EXA2PRO: A Framework for High Development Productivity on Heterogeneous Computing Systems
Lazaros Papadopoulos, Dimitrios Soudris, Christoph Kessler, August Ernstsson, Johan Ahlqvist, Nikos Vasilas, Athanasios I Papadopoulos, Panos Seferlis, Charles Prouveur, Matthieu Haefele, et al.

To cite this version:
Lazaros Papadopoulos, Dimitrios Soudris, Christoph Kessler, August Ernstsson, Johan Ahlqvist, et al.. EXA2PRO: A Framework for High Development Productivity on Heterogeneous Computing Systems. IEEE Transactions on Parallel and Distributed Systems, 2021, Special Section on Innovative R&D toward the Exascale Era, 10.1109/TPDS.2021.3104257 . hal-03318644

HAL Id: hal-03318644
https://inria.hal.science/hal-03318644
Submitted on 10 Aug 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
EXA2PRO: A Framework for High Development Productivity on Heterogeneous Computing Systems

Lazaros Papadopoulos, Dimitrios Soudris, Christoph Kessler, August Ernsthsson, Johan Ahlqvist, Nikos Vasilas, Athanasios I. Papadopoulos, Panos Seferlis, Charles Prouveur, Matthieu Haefele, Samuel Thibault, Athanasios Salamanis, Theodoros Ioakimidis, Dionyios Kehagias

Abstract—Programming upcoming exascale computing systems is expected to be a major challenge. New programming models are required to improve programmability, by hiding the complexity of these systems from application developers. The EXA2PRO programming framework aims at improving developers’ productivity for applications that target heterogeneous computing systems. It is based on advanced programming models and abstractions that encapsulate low-level platform-specific optimizations and it is supported by a runtime that handles application deployment on heterogeneous nodes. It supports a wide variety of platforms and accelerators (CPU, GPU, FPGA-based Data-Flow Engines), allowing developers to efficiently exploit heterogeneous computing systems, thus enabling more HPC applications to reach exascale computing. The EXA2PRO framework was evaluated using four HPC applications from different domains. By applying the EXA2PRO framework, the applications were automatically deployed and evaluated on a variety of computing architectures, enabling developers to obtain performance results on accelerators, test scalability on MPI clusters and productively investigate the degree by which each application can efficiently use different types of hardware resources.

Index Terms—programming models, skeleton programming, task-based runtime systems, programming productivity, heterogeneous systems, exascale computing

I. INTRODUCTION

The building of exascale computing systems is a huge investment both financially and in terms of scientific effort. Therefore, it is beneficial to enable more applications to access these systems, or at least, promote their transition towards the exascale computing direction. Towards this end, advanced programming models and environments are needed that will assist developers in addressing critical challenges in terms of efficient application deployment, such as maintaining development productivity in the heterogeneous exascale computing environments.

EXA2PRO is a framework developed in the context of the EXA2PRO H2020 EU project [1] and its goal is to allow the transition of more HPC applications to exascale computing, by focusing on development productivity. More specifically, the EXA2PRO framework allows developers to evaluate applications across a variety of different architectures and therefore, to efficiently exploit accelerators, such as GPUs and FPGAs. Thus, EXA2PRO lowers the barrier of access and enables more HPC applications to exploit the exascale computing systems, by hiding their complexity from application developers.

The EXA2PRO framework combines the skeleton programming model with a multi-backend approach to enable portability and programmability across a variety of architectures. It provides high-level software abstractions (skeletons, multi-variant components), through the SkePU tool, that implement data parallel computational patterns commonly found in HPC applications (map, mapreduce, stencil, etc.), which are automatically generated for various backends (OpenMP, CUDA, OpenCL, etc.). The framework encapsulates all low-level tuning and optimization tasks to allow developers to focus on application programming and to automatically evaluate applications across different architectures, instead of manually optimizing and porting to accelerators, thus greatly increasing development productivity. Additionally, the EXA2PRO framework makes advanced programming models more accessible to application developers. EXA2PRO supports the data-flow programming model and MPI through its runtime system, allowing developers to efficiently exploit large-scale heterogeneous computing systems, by hiding their complexity. More specifically, the EXA2PRO framework integrates the StarPU runtime system, which can be used either as another EXA2PRO backend or it can be used in cases in which the skeletons may not fit the overall software design of the application and StarPU task-based programming model maybe simpler to apply (even if much more involved in terms of lines of code and expressiveness complexity).

The purpose of this work is to highlight the key features of the EXA2PRO framework and to evaluate and demonstrate its development productivity features, based on a set of scientific applications that need to be evaluated in more complex architectures and target their transition to exascale computing. The main contributions are the following:

- We present the EXA2PRO framework, which enables...
the evaluation of applications in heterogeneous exascale computing systems \(^1\).

- We demonstrate the use of EXA2PRO in a variety of applications, which target the transition to exascale computing systems and show how the multi-backend feature of EXA2PRO can improve the development productivity.

The rest of the paper is organized as follows: Section II describes the related work. The EXA2PRO framework and its components are presented in Section III. The evaluation results are analyzed and discussed in Section IV. Finally, in Section V we draw conclusions.

II. RELATED WORK

The EPEEC programming framework has similar goals as EXA2PRO [2], which also targets heterogeneous exascale supercomputers. A programming framework, developed in the context of the EPIGRAM-HS project, addresses heterogeneity challenges both in terms of acceleration (GPUs, FPGAs) and in terms of memory (HBM, NVM) [3]. The ASPIDE project develops a programming framework tailored to Big Data applications deployed on exascale computing systems [4]. Programming models and runtimes for exascale are developed in the context of the US Exascale Computing Project, such as enhancements of the MPI standard for a task-based programming model [5]. Kokkos provides high-level abstractions for parallel loop executions, which are mapped to a runtime for achieving performance across various architectures [6]. Although the above libraries and frameworks have similar goals, they focus on different aspects of the programmability of the exascale computing systems, utilize different programming models and attempt to address different exascale computing challenges.

The EXA2PRO framework focuses on the programmability of heterogeneous systems and integrates programming models that are different from the aforementioned models: skeleton programming in EXA2PRO is used to implement data-parallel patterns and the data-flow programming model is used to support FPGA-based data-flow engines (DFE) [7]. The goal is to enable the transition of more HPC applications towards the exascale computing direction.

OpenACC is an API that targets programmability and portability across diverse architectures and it is based on compiler directives [8]. OpenACC, as well as SYCL and other programming models can be considered complementary to EXA2PRO. However, SkePU skeletons are more high-level and more powerful, as their implementations are not constrained by the given loop nest structure in an OpenACC source program.

Among libraries exposing a task-based programming model, OmpSs [9] PaRSEC [10] and DTD (Dynamic Task Discovery) also provide support for heterogeneous accelerator-based and distributed systems, and support for dynamic task scheduling to some extent. In comparison, the EXA2PRO runtime, StarPU [11], reaches more advanced task scheduling, by achieving compromises between task acceleration, data transfer cost, and energy cost. Its pluggable task scheduling interface allows to integrate state-of-the-art heuristics (e.g. HEFT, HeteroPrio).

III. EXA2PRO FRAMEWORK

The EXA2PRO framework is based on a set of advanced programming models, supported by a toolflow that allows developers to evaluate applications across a variety of architectures and to efficiently exploit accelerators, such as GPUs and FPGAs, thus supporting the transition of applications to exascale computing through increased development productivity. Its software stack is shown in Fig. 1. The EXA2PRO high-level programming model is based on a set of skeletons, multi-variant components, smart containers and other APIs. To apply the EXA2PRO framework to an application, developers are expected to replace the algorithmic pattern of the application with the EXA2PRO high-level programming interface.

The most important features of the EXA2PRO framework are the following:

- It provides a simple sequential API for several common parallel algorithmic patterns
- The EXA2PRO tools automatically generate optimized and tunable implementations for a variety of heterogeneous systems (implementation variants). Users may provide custom implementations (multi-variant components).
- Suboptimal variants are manually or automatically discarded, while the ones expected to provide better results are forwarded to the EXA2PRO runtime system
- The EXA2PRO runtime system provides optimized variant selection, scheduling and data transfers to accelerators

The EXA2PRO framework hides the complexity and the low-level implementation details of large-scale heterogeneous systems from application developers. It encapsulates accelerator code optimizations, scheduling decisions, synchronization issues, data transfer optimization and other low-level tasks. Thus, by providing high-level software abstractions for parallel patterns widely used in HPC applications, the EXA2PRO framework gives the opportunity to more applications to access complex large-scale computing systems.

---

1 A docker and a singularity container of the framework are available on the EXA2PRO website: https://exa2pro.eu/#developers
A. EXA2PRO High-Level Programming Model

At the top level, an EXA2PRO application consists of zero or more multi-variant components, i.e., modularized functions considered for possible execution on an accelerator, and of outer code (Fig. 1). Outer code is CPU code which might also contain explicit parallelization constructs (e.g., MPI) and skeleton calls, see Section III-B. The components may have different, user-provided implementation variants for different programming and execution platforms in a heterogeneous system (e.g., OpenMP for CPU, CUDA and OpenCL for GPU, etc.), i.e., inner code, and are annotated with metadata e.g. about their dependencies and deployment. The EXA2PRO framework synthesizes an executable application from the components and outer code, which will be free to select among implementation variants at runtime (see the selection transitions in the "gearbox" in Fig. 2), depending on the current execution context (such as operand sizes and locations), and supported by the EXA2PRO runtime system. Also skeleton calls allow to select statically or (automatically) at runtime\(^2\) among their implementations, which are provided by the SkePU skeleton framework instead.

The SkePU programming interface and implementation is inherently based on modern C++, especially on template metaprogramming and variadic templates, and user functions must be written in C++. But SkePU also offers a prototype FORTRAN API for access to SkePU data-containers from FORTRAN code, and the SkePU pre-compiler can automatically generate FORTRAN wrapper functions for skeleton instance invocation.

B. SkePU

Skeletons are closely related to so-called (parallel) algorithmic patterns [14]; in fact, skeletons implement patterns. Multi-backend skeletons are portable skeletons that realize data-parallel patterns such as map, reduce, scan, stencil etc., and that have internally optimized implementations for different resources (CPU, GPU, ...) of heterogeneous parallel systems.

2In node-level SkePU, dynamic selection of the expected best implementation based on skeleton call operand sizes and locations can be done in SkePU itself after off-line training [12], [13]; in cluster-level SkePU and for multi-variant components, automated performance modeling and selection are delegated to the runtime system (Sect. III-E).

In EXA2PRO, we developed the third generation of the skeleton programming framework SkePU [15], which is available both stand-alone as SkePU-3 [16] but also integrated into the EXA2PRO high-level programming model. Compared to SkePU-2 [17], new skeletons and data-container types have been added, the programming interface has been streamlined and the memory consistency model for the smart data-containers has been relaxed. SkePU-3 is available as open-source under a modified BSD license [16].

SkePU [15], [16] provides a single, sequential-like programming interface extending modern C++ with data-parallel skeletons, see Tables I–II. For each skeleton, implementation variants (backends) for sequential C++, OpenMP, CUDA (for Nvidia GPUs) and OpenCL (for all GPUs) are available, and also multi-GPU and hybrid CPU-GPU code can be selected [18]. In EXA2PRO, an additional backend for the MPI bindings of StarPU has been developed. (Access to further platforms such as FPGA accelerator cards can be achieved indirectly via SkePUs interoperability with multi-variant components, which are described in Section III-C.)

<table>
<thead>
<tr>
<th>Skeleton</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>Element-wise 1D–4D transformation of arbitrary arity</td>
</tr>
<tr>
<td>MapPairs</td>
<td>Generic 2D Cartesian-product style computation, arbitrary arity</td>
</tr>
<tr>
<td>MapOverlap</td>
<td>1D–4D convolution with arbitrary per-dim. stencil radius</td>
</tr>
<tr>
<td>Reduce</td>
<td>Generic reduction</td>
</tr>
<tr>
<td>Scan</td>
<td>Generic prefix-sums computation</td>
</tr>
<tr>
<td>MapReduce</td>
<td>Fusion of Map and Reduce</td>
</tr>
<tr>
<td>MapPairsReduce</td>
<td>Fusion of MapPairs and row-wise or column-wise Reduce</td>
</tr>
</tbody>
</table>

Table I gives short descriptions of the current data-parallel skeletons in SkePU, and the interface properties (such as supported arity, dimensionality of operands and special operand access patterns) are summarized in Table II; for detailed descriptions and illustrations for all skeletons we refer to [15]. In a co-design effort, based on feedback by project partners developing and porting HPC applications to the new programming model, SkePU has been extended with new skeletons such as MapPairs (generalized Cartesian-product style computation pattern, supporting a variadic number of vector operands in each dimension) and MapPairsReduce (fused MapPairs with subsequent row-wise or column-wise reduction), and its programming interface has been generalized and streamlined, such as multi-valued return from all Map-based skeletons and user functions (as known from Python) and a streamlined interface for the MapOverlap (stencil pattern) skeleton. Minor improvements include optional dynamic scheduling in the OpenMP backend of most skeletons.

User functions are problem-specific functions that can be plugged as code parameters into skeletons at instantiation. They must be side-effect-free C++ functions with restricted C++ features (in fact, they are basically C functions with container access proxy objects (see Sec. III-D and the example in Fig. 3) because they must be translatable into platform-specific code with limited or no C++ support, in particular for OpenCL. For the same reason, library functions called from user functions must be portable, i.e., exist on all supported target platforms, or explicitly implemented as user functions too.
In-line definition of user functions in skeleton instantiations using C++ lambda functions is supported.

Cluster support in SkePU comes in two different (mutually exclusive) flavors:

- **Outer MPI** (also known as MPI+SkePU) applications use SkePU only at node-level and let the programmer use ordinary MPI calls for explicit inter-node communication, duly framed by `skepu::external` guards (explained later in this paragraph) to preserve sequential semantics and guarantee memory consistency for communicated operands that are managed by SkePU data-containers. For skeleton calls in outer MPI code, the MPI backend of the skeleton is disabled.

- **Inner MPI** applications have MPI-free, plain sequential-looking SkePU source code where all cluster-level parallelism is exploited by the SkePU MPI backend working on distributed data-containers, hence all communication is implicit.

As an illustrative example, Fig. 3 shows an excerpt of a brain modeling simulation code in SkePU (see also Sect. IV-C). It is a time-driven simulator, in which each neuron updates its current state. The user function `SimulateNeuron_Skepu` is used in an instantiation (line 29) of the Map skeleton, here with no (`<>`) element-wise accessed operands. The user function takes as its main operands a weight matrix `m` and the `state` vector, prepared for random access with the `Mat` access proxy (alternatively, row-wide random access with `MatRow`), resp. `Vec` access proxy in the user function; see Sect. III-D for SkePU data-container and access proxy types).

The skeleton instance is called in line 48. By default, SkePU selects the most parallel backend, but here the user explicitly prescribes a backend (line 31) passed in as a command-line argument string, which is set in the `spec` object attached to the skeleton instance `SimulateTimestepPerNeuron`. Alternatively, the selection autotuning mechanism of SkePU could be used.

As illustrated in the example code snippet in Fig. 3, SkePU programs have three different code scopes: managed scope (user functions, with restricted functionality due to strong portability requirements), unmanaged scope (code outside `skepu::external`, where data-containers can be defined and skeletons be instantiated and called), and external scope (guarded code within `skepu::external`. `skepu::external` is used to frame I/O and MPI operations

### Table II

<table>
<thead>
<tr>
<th>Skeleton</th>
<th>Out Arg Arity / Dim</th>
<th>El.-wise In Arg Arity / Dim</th>
<th>Proxy In Args</th>
<th>Uniform In Args</th>
<th>Index Avail</th>
<th>Special Access</th>
<th>Proxied Supports</th>
<th>Node-Level SkePU Backends</th>
<th>Cluster-Level SkePU Backends</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>Variadic / 1D-4D</td>
<td>Variadic / 1D-4D</td>
<td>Variadic</td>
<td>Variadic</td>
<td>Yes</td>
<td>MatRow, MatCol</td>
<td>•</td>
<td>OpenMP</td>
<td>CUDA</td>
</tr>
<tr>
<td>MapVars</td>
<td>Variadic / 1D-4D</td>
<td>Variadic / 1D-4D</td>
<td>Variadic</td>
<td>Variadic</td>
<td>Yes</td>
<td>MatRow, MatCol</td>
<td>•</td>
<td>OpenMP</td>
<td>CUDA</td>
</tr>
<tr>
<td>MapOverlap</td>
<td>1 / 1D-4D</td>
<td>1 / 1D-4D</td>
<td>Variadic</td>
<td>Variadic</td>
<td>Yes</td>
<td>Region</td>
<td>•</td>
<td>OpenMP</td>
<td>CUDA</td>
</tr>
<tr>
<td>Reduce</td>
<td>1 / 0D-1D</td>
<td>1 / 1D</td>
<td>Variadic</td>
<td>Variadic</td>
<td>Yes</td>
<td>MatRow, MatCol</td>
<td>•</td>
<td>OpenMP</td>
<td>CUDA</td>
</tr>
<tr>
<td>MapReduce</td>
<td>Variadic / 0D</td>
<td>Variadic / 1D-4D</td>
<td>Variadic</td>
<td>Variadic</td>
<td>Yes</td>
<td>MatRow, MatCol</td>
<td>•</td>
<td>OpenMP</td>
<td>CUDA</td>
</tr>
<tr>
<td>MapFailsReduce</td>
<td>Variadic / 1D</td>
<td>1 / 1D (x2) / 1D</td>
<td>Variadic</td>
<td>Variadic</td>
<td>Yes</td>
<td>•</td>
<td>•</td>
<td>OpenMP</td>
<td>CUDA</td>
</tr>
</tbody>
</table>

Fig. 3. Excerpt of the brain modeling simulation code in SkePU.
access data managed by data-containers to make sure that, given the weak memory consistency model of SkePU-3 in non-managed code, data to be communicated is valid in main memory. The last part in Fig. 3 (lines 54–61) illustrates this: For the data-container list in the optional read argument of skepu::external, SkePU makes sure to flush any possibly software-cached elements of these data-containers back to main memory, as some may still reside (only) in device memory after being written by a skeleton call (which is in general, such as here, statically unknown). For details we refer to [15].

Where a skeleton does not fit well, e.g., for a parallel loop with irregular, indirect access pattern through a permutation index vector, an overly general skeleton such as a Map<0> with an access proxy for a random-access operand instead of an element-wise (or, e.g., row-wise) accessed operand has to be used, which leads to higher communication volume for distributed memory (GPU, cluster) execution and also prevents execution on accelerators where the indirectly accessed data-container does not fit in device memory in its entirety.

C. Multi-Variant Components

Multi-variant components are modularized, user-provided functions that may have multiple, user-provided implementation variants written using different programming models such as C++, OpenMP, CUDA, OpenCL or MAXI (i.e. the programming model of DFEs), and that are annotated with metadata in an XML based annotation framework to support target platform-specific code generation and deployment. EXA2PRO extends this concept from earlier work [19] by adding new target platform types (MPI, Maxeler DFE) and meta-modeling new function properties such as operand access patterns. The deployment toolchain for this is called ComPU.

EXA2PRO multi-variant components are interoperable with SkePU skeleton instances as they work on the same data-container types (Sec. III-D), see also Fig. 4 for an illustration of SkePU-ComPU interoperability, and provide a flexible escape mechanism for cases where SkePU skeletons do not fit (well) a computation’s structure.

D. Smart data-containers and access proxies

Smart data-containers are high-level STL-based generic abstractions for array-based data structures passed as operands to and from calls to skeletons and multi-variant components. We call them “smart” as they can transparently perform runtime optimizations for data transfer [20], [21], data locality [22] and memory management. In EXA2PRO (and SkePU-3), currently four such data-containers are supported for 1D to 4D data abstractions: Vector<> (generic 1D vector type similar to STL std::vector), Matrix<> (generic dense matrix), Tensor3<> and Tensor4<>() (generic 3D and 4D dense tensor, respectively).

In addition, there are a number of different container access proxies that are to be used in managed scope and that encode different access patterns, which help SkePU to reduce communication effort. For example, the most generic Mat<...> container access proxy allows for random access to any element of a Matrix<...> operand passed to a skeleton instance call, while the more specific MatRow<...> and MatCol<...> access proxies allow random access only within one row or column of a Matrix<...> operand respectively. For pure element-wise access, no container proxy is needed at all on the user function side. For details see [15].

E. Runtime System: StarPU

EXA2PRO leverages the StarPU [11] task-based runtime system. StarPU aims at providing optimized application execution over large clusters of heterogeneous systems, such as composed of CPUs, GPUs and FPGAs. It uses a task-based programming paradigm which captures high-level information from the application, and allows its scheduler to be very well informed of the computation performed by the application.

The fundamental model of the StarPU runtime system is the submission of a Directed Acyclic Graph (DAG) of tasks. This is expressed thanks to the notions of data, codelet, and tasks. The dependencies between tasks can be set explicitly, or inferred from the data dependencies for the convenience of the programmer.

In the context of the EXA2PRO framework, StarPU is used as the execution backend for the composition framework. SkePU automatically generates, from the skeleton-based application, a source code that leverages the StarPU interface to submit a task graph. For each task, it provides StarPU with the different implementation variants for the different architectures available on the target platform (CPU, GPU, DFE, ...), by gathering them in a codelet (i.e. the collection of functions which achieve the same computation, thus various implementations variants for the computation). Similarly, the different implementation variants provided by the application in the multi-variant components are exposed in StarPU codelets.

From the description of the task graph by SkePU and the composition framework, the StarPU runtime system orchestrates the execution of tasks among the different execution units and memory locations. Since it has a complete vision over the data and the tasks to be computed, it is able to provide the application with a flurry of features without further effort from the application programmer, such as optimized task scheduling or data transfer overlapping.

IV. EVALUATION

The EXA2PRO framework was applied to four HPC applications: CO2 capture process design and control, a supercapacitor simulator (Metalwalls), a neural simulator (brain modeling) and an Ordinary Differential Equations (ODE) solver.
TABLE III
APPLICATIONS USED TO EVALUATE THE EXA2PRO FRAMEWORK AND COMPUTING SYSTEMS SPECIFICATIONS

<table>
<thead>
<tr>
<th>Application</th>
<th>Motivation</th>
<th>EXA2PRO tool(s) applied</th>
<th>Metrics</th>
<th>Computing Systems CPU/GPU/DFE Specs</th>
</tr>
</thead>
<tbody>
<tr>
<td>CO₂ capture</td>
<td>(i) Parallelize calculations of CO₂ capture process model</td>
<td>(i) SkePU: Map skeleton</td>
<td>(i) Speedup on single node compared to original</td>
<td>(i) Local cluster: Xeon E-2176G0 (4 cores @1.8GHz), NVIDIA Quadro P620</td>
</tr>
<tr>
<td></td>
<td>(ii) Evaluate OpenMP, CUDA and StarPU-MPI</td>
<td>(ii) StarPU: (through SkePU StarPU-MPI support)</td>
<td>(ii) Scalability</td>
<td>(ii) ARIS: Xeon E5-2680v2 (20 cores @2.8GHz)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(iii) Piz Daint: Xeon E5-2690v3 (12 cores @2.6GHz), NVIDIA Tesla P100</td>
</tr>
<tr>
<td>Metalwalls</td>
<td>(i) Evaluate SkePU, StarPU and DFE implementations</td>
<td>SkePU: MapPairsReduce StarPU</td>
<td>Speedup, scalability and Perf/Watt compared to original</td>
<td>(i) Intel Haswell E5-2660v3 (12 cores @2.6GHz), Maxeler DFE Max5C</td>
</tr>
<tr>
<td></td>
<td>(ii) Evaluate scalability through SkePU, StarPU</td>
<td></td>
<td></td>
<td>(ii) Piz Daint: Xeon E5-2690v3 (12 cores @2.6GHz), NVIDIA Tesla P100</td>
</tr>
<tr>
<td>Brain Modeling</td>
<td>Evaluate OpenMP, GPU and StarPU-MPI implementations</td>
<td>SkePU: Map skeleton with MatRow access proxy</td>
<td>(i) Execution time on single node compared to original</td>
<td>(i) Local Cluster: 2x Intel Xeon Gold 6138 (40 cores @2GHz) NVIDIA Tesla V100 GPU</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(ii) Scalability</td>
<td>(ii) Tetralith cluster: 2x Xeon Gold 6130 (2x16 cores @2.1GHz)</td>
</tr>
<tr>
<td>ODE solver</td>
<td>Evaluate SkePU with an iterative data-parallel application with a long dependence chain</td>
<td>SkePU: Map, Reduce, MapReduce, MapOverlap</td>
<td>(i) Speedup on single node</td>
<td>Tetralith cluster: 2x Xeon Gold 6130 (2x16 cores @2.1GHz)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(ii) Scalability</td>
<td></td>
</tr>
</tbody>
</table>

applications were selected based on the following two criteria: i) To belong to different application domains ii) To be HPC applications currently deployed in HPC clusters, which aim at evaluation on heterogeneous large-scale computing systems. Specific tools of the EXA2PRO framework were applied to each application, based on each application requirements, as shown on Table III.

A. CO₂ capture process design and control

This application pertains to the design of efficient and robust CO₂ capture systems. The employed CO₂ capture process models and overall design optimization approach are described in [23]. The goal was to test for the first time advanced programming models, to evaluate the application on heterogeneous systems and large number of nodes and to assess its scalability and speedup. The size of the solved problems ranges from $4 \times 10^4$ to $4 \times 10^6$ algebraic equations, whereas tests are performed in up to 1000 CPU threads and to GPUs in a local cluster, the ARIS and Piz Daint supercomputers. The Map skeleton of SkePU is used as illustrated in Fig. 5, where it is compared to the original, sequential code.

In the CO₂ capture case, the core operation ported to SkePU is represented by the non-linear system of algebraic equations that are used to model the chemical process system [23]. The ported functions are used in a sequence of two calculation stages. In the first stage, the functions are used within a constrained optimization problem formulation, to determine the optimum solution under steady-state operation. In the second stage, the Karush-Kuhn-Tucker optimality conditions [24] are used to transform the problem into an equivalent algebraic formulation [25]. This is used as part of a homotopy-continuation method that employs a predictor-corrector numerical technique to identify an optimum solution under external variability, within a control structure. In both stages, the equality and inequality constraints of the constrained optimization problem and their gradients are implemented in SkePU. The algorithms used in the two stages are IPOPT [26] and PITCON [27]. The former calls the functions in SkePU to calculate the Hessian and to perform the Newton step for the solution of the nonlinear equations.

Fig. 6(a) depicts the speedup based on the total execution time, using SkePU parallel backends over single-core execution, to evaluate the CO₂ capture model for an increasing model size. The total execution time comprises the times for the following tasks: (a) The time to create-allocate a SkePU vector to store the output. (b) The call to the SkePU skeleton (c) The SkePU flush operation to make sure that the vector’s elements are valid in main memory. In contrast with (a) and (b), this task has an impact only when a GPU backend is used.

Fig. 6(b) shows the time performance of each task that makes up the total time reported in Fig. 6(a). The total time used in the speedup bar for OpenMP is the sum of the time for the OpenMP call task (orange line) and the time for the SkePU vector create task (blue line). The total time used in the speedup bar for CUDA is the sum of the time for the CUDA call task (green line), the time for the CUDA flush task (red line) and the time for the SkePU vector create task (blue line). The OpenCL case is not discussed as the performance is similar to that of CUDA.

In the OpenMP case, when the mathematical model comprises fewer equations, the total time is mainly due to the OpenMP call task (orange line), with the SkePU vector create task having a much smaller contribution. When the number of equations increases, the OpenMP call task is still slower but the vector create task has a significant contribution to the overall time. This behavior leads to an increase in the total time, which is reflected in the reduction of the speedup shown in Fig. 6(a). In the CUDA case, it appears from Fig. 6(b) that for fewer equations the total time is dominated by the CUDA call task (green line). As the number of equations increases, there is an intense change (increase) in the slope of the CUDA skeleton call task (green line), indicating the additional effort needed to solve a problem of larger size. The similar execution time between CUDA and OpenMP at or above $400 \times 10^3$ equations, is due to the very efficient CPU of the local cluster, as opposed to its less efficient GPU (see also Fig. 7(c)). The speedup is considerably higher for a larger number of equations, because the larger problem size is executed considerably faster in the GPU, compared to the single CPU. This is despite that the actual CUDA call time increases as the number of equations increases.
Fig. 6. CO$_2$ capture: (a) Speedup using SkePU for various backends on a single node, considering all tasks of Fig. 5. (b) Break down of time for tasks create of SkePU vector, call with OpenMP, call with CUDA and flush with CUDA. (Results on Local Cluster (single node) for both figures).

Fig. 7. CO$_2$ capture: (a) Speedup when only the call task is considered in all cases. (Local Cluster) (b) Speedup for increasing number of equations when only the call task is considered (Piz Daint). (c) Execution time on CPUs and GPUs of Piz Daint and Local Cluster (d) Execution time on ARIS using SkePU-StarPU MPI backend for multiple MPI nodes.

Fig. 5. Original and EXA2PRO CO$_2$ capture code.

of cores increases, and a fixed workload is distributed to the parallel cores, the task size per core decreases. In this respect, the application does not scale above 200 cores, since the task size becomes lower than 1ms and the runtime overhead per task, as explained in earlier works [28], has significant impact on the execution time. Therefore, even if we run experiments above 1000 cores, no speedup is expected. The model of the CO$_2$ capture process is flexible and enables the representation of phenomena at different levels of fidelity. The considered range of equations represents typical models of medium to higher fidelity, exhibiting significant speedup and scalability compared to the original implementation.

The effort of applying SkePU to the CO$_2$ capture was performed by a senior engineer, already familiar with the application, that required about 2 weeks of training on EXA2PRO and 4 weeks of development.

B. Supercapacitor simulation

Metalwalls is a classical molecular dynamic code dedicated to the accurate simulation of electrochemical systems like supercapacitors, devices able to store energy under electrostatic form. The typical simulated system is made of two carbon
electrodes immersed in an ionic liquid. At each time step, the simulation computes with a matrix-free conjugate gradient the charge density on electrodes such that they conserve a constant potential which is the heart of the mini-app studied in this paper. Details on the code and references to the model can be found in [29].

The motivation of using EXA2PRO for this application is to evaluate the programming effort to rewrite its compute intensive part and measure to which extent the original good performance can be retrieved with SkePU, StarPU and MaxJ (i.e. the programming model of DFEs).

The original code is written in Fortran 90 and is parallelized with MPI. With the introduction of OpenACC directives, a single GPU version is also available. Fig. 8 shows how its serial version has been rewritten in the SkePU framework with a single MapPairReduce. The code complexity is similar to serial code while enabling its parallel execution on CPU and GPU. In order to maximize data locality, the original implementation computes by blocks the contribution, of each atom pairs, to the electrostatic potential \( V \), instead of spanning the whole atoms array on both loops as shown here. Blocks are then distributed among the different MPI ranks to enable parallel computations on cores and nodes. As there is no data distribution, all ranks have all data and a single MPI_Allreduce aggregates all contributions and updates all ranks. The StarPU porting introduces a shared memory level into the application allowing the static distribution within MPI to balance efficiently the load. As a result, the application does not benefit so much from StarPU. The relative small tasks size submitted amplifies this overhead and makes FPGA a relevant device for the exascale era. Fig. 9 shows the speedup in execution time of the different programming models on different architectures. The native StarPU implementation outperforms the original MPI by only 10%. The expected gains coming from the shared memory level introduced are consumed by the scheduler overhead. The relative small tasks size submitted amplifies this overhead and the regular behaviour of the application allows the static distribution within MPI to balance efficiently the load. As a result, the application does not benefit so much from StarPU. The SkePU implementation degrades performance by nearly a factor three. Data locality of the generated implementation cannot compete yet with the original optimised version. The DFE implementation is 20x faster than the original CPU version but 2x slower than the original GPU version. This very large discrepancy with CPU can be explained by the fact that the application is mostly compute bound. Thus, the computing power of both devices can be efficiently leveraged and time to solution of the DFE implementation is behind the GPU but still competitive.

These implementations have been evaluated on the test-case used in production in [30] that contains 42490 electrode atoms. In the context of material science for supercapacitors, this is a large test case. It is however small considering computing platform capabilities and is thus a limiting factor for extreme scalability.

Fig. 9 top left shows the speedup in execution time of the different programming models on different architectures. The native StarPU implementation outperforms the original MPI by only 10%. The expected gains coming from the shared memory level introduced are consumed by the scheduler overhead. The relative small tasks size submitted amplifies this overhead and the regular behaviour of the application allows the static distribution within MPI to balance efficiently the load. As a result, the application does not benefit so much from StarPU. The SkePU implementation degrades performance by nearly a factor three. Data locality of the generated implementation cannot compete yet with the original optimised version. The DFE implementation is 20x faster than the original CPU version but 2x slower than the original GPU version. This very large discrepancy with CPU can be explained by the fact that the application is mostly compute bound. Thus, the computing power of both devices can be efficiently leveraged and time to solution of the DFE implementation is behind the GPU but still competitive.

Fig. 9 top left shows the speedup in execution time of the different programming models on different architectures. The native StarPU implementation outperforms the original MPI by only 10%. The expected gains coming from the shared memory level introduced are consumed by the scheduler overhead. The relative small tasks size submitted amplifies this overhead and the regular behaviour of the application allows the static distribution within MPI to balance efficiently the load. As a result, the application does not benefit so much from StarPU. The SkePU implementation degrades performance by nearly a factor three. Data locality of the generated implementation cannot compete yet with the original optimised version. The DFE implementation is 20x faster than the original CPU version but 2x slower than the original GPU version. This very large discrepancy with CPU can be explained by the fact that the application is mostly compute bound. Thus, the computing power of both devices can be efficiently leveraged and time to solution of the DFE implementation is behind the GPU but still competitive.

Fig. 9 shows the speedup in execution time of the different programming models on different architectures. The native StarPU implementation outperforms the original MPI by only 10%. The expected gains coming from the shared memory level introduced are consumed by the scheduler overhead. The relative small tasks size submitted amplifies this overhead and the regular behaviour of the application allows the static distribution within MPI to balance efficiently the load. As a result, the application does not benefit so much from StarPU. The SkePU implementation degrades performance by nearly a factor three. Data locality of the generated implementation cannot compete yet with the original optimised version. The DFE implementation is 20x faster than the original CPU version but 2x slower than the original GPU version. This very large discrepancy with CPU can be explained by the fact that the application is mostly compute bound. Thus, the computing power of both devices can be efficiently leveraged and time to solution of the DFE implementation is behind the GPU but still competitive.

These implementations have been evaluated on the test-case used in production in [30] that contains 42490 electrode atoms. In the context of material science for supercapacitors, this is a large test case. It is however small considering computing platform capabilities and is thus a limiting factor for extreme scalability.

Fig. 9 top left shows the speedup in execution time of the different programming models on different architectures. The native StarPU implementation outperforms the original MPI by only 10%. The expected gains coming from the shared memory level introduced are consumed by the scheduler overhead. The relative small tasks size submitted amplifies this overhead and the regular behaviour of the application allows the static distribution within MPI to balance efficiently the load. As a result, the application does not benefit so much from StarPU. The SkePU implementation degrades performance by nearly a factor three. Data locality of the generated implementation cannot compete yet with the original optimised version. The DFE implementation is 20x faster than the original CPU version but 2x slower than the original GPU version. This very large discrepancy with CPU can be explained by the fact that the application is mostly compute bound. Thus, the computing power of both devices can be efficiently leveraged and time to solution of the DFE implementation is behind the GPU but still competitive.

Fig. 9 shows the speedup in execution time of the different programming models on different architectures. The native StarPU implementation outperforms the original MPI by only 10%. The expected gains coming from the shared memory level introduced are consumed by the scheduler overhead. The relative small tasks size submitted amplifies this overhead and the regular behaviour of the application allows the static distribution within MPI to balance efficiently the load. As a result, the application does not benefit so much from StarPU. The SkePU implementation degrades performance by nearly a factor three. Data locality of the generated implementation cannot compete yet with the original optimised version. The DFE implementation is 20x faster than the original CPU version but 2x slower than the original GPU version. This very large discrepancy with CPU can be explained by the fact that the application is mostly compute bound. Thus, the computing power of both devices can be efficiently leveraged and time to solution of the DFE implementation is behind the GPU but still competitive.

These implementations have been evaluated on the test-case used in production in [30] that contains 42490 electrode atoms. In the context of material science for supercapacitors, this is a large test case. It is however small considering computing platform capabilities and is thus a limiting factor for extreme scalability.

Fig. 9 top left shows the speedup in execution time of the different programming models on different architectures. The native StarPU implementation outperforms the original MPI by only 10%. The expected gains coming from the shared memory level introduced are consumed by the scheduler overhead. The relative small tasks size submitted amplifies this overhead and the regular behaviour of the application allows the static distribution within MPI to balance efficiently the load. As a result, the application does not benefit so much from StarPU. The SkePU implementation degrades performance by nearly a factor three. Data locality of the generated implementation cannot compete yet with the original optimised version. The DFE implementation is 20x faster than the original CPU version but 2x slower than the original GPU version. This very large discrepancy with CPU can be explained by the fact that the application is mostly compute bound. Thus, the computing power of both devices can be efficiently leveraged and time to solution of the DFE implementation is behind the GPU but still competitive.
weeks. The resulting code base of the StarPU implementation (650 additional LOC) has a higher complexity compared to the original MPI implementation, while the SkePU one (450 specific LOC that replaces original kernels) is much simpler and slightly shorter as it is very close to a serial implementation. Earlier works that evaluate the complexity of StarPU implementations can be found in the literature [28]. The DFE porting with MaxJ took however between eight and twelve months. The differences in approach and learning environment are the main reasons for this large gap and the resulting implementation (925 specific LOC that replaces original kernels) is even more complex than the original MPI one. Despite such a discrepancy, the time needed to obtain an efficient FPGA implementation is still much lower compared to the development of a low level VHDL implementation.

C. Brain Modeling

The Brain Modeling application is a time-driven simulator of biophysically detailed, Extended Hodgkin-Huxley (eHH) models of individual neurons [31] and supports the entire NeuroML standard [32]. It is a ground-up new design that focuses on supporting large-scale networks on HPC infrastructures. It can be considered an electrical circuit, in which the non-linear dynamics require step-by-step simulation and the continuous interaction between the neurons requires communication at each step. Therefore, at each step, for each neuron the following are applied: i) Accumulation of interaction of each neuron with its adjacent neurons (coupling strength between each neuron pair is defined in a weight matrix) and ii) Simulation progress considering internal dynamics and inter-neuron currents.

We applied SkePU in the brain modeling application, to target multiple parallelization platforms, including single-node OpenMP and OpenCL GPU, and multi-node MPI+OpenMP, without substantial changes to the original algorithm. Since the acceleration platforms incur additional development overhead, the EXA2PRO porting features enable rapid evaluation of the application on different architectures.

The Map skeleton uses an input data container, to produce an output element for each one in the input containers, based on a user function. This skeleton fit well with the main computational pattern of the brain modeling application, because within each simulation step, each neuron’s state can be advanced independently. The memory pattern is similar to matrix-vector multiplication. The user function provided to the Map skeleton produces each neuron’s updated state, using present data and a weight matrix.

A code snippet about how the Map skeleton was applied to the original brain modeling application, is shown in Fig. 3. The original application code is OpenMP-enabled and was used as performance baseline. The derived SkePU implementations were evaluated in terms of performance and scalability.

Evaluation results of SkePU GPU and SkePU OpenMP under 80 threads, on a single node (Intel Xeon with 2x20 H/T cores) are shown in Fig. 10. SkePU GPU implementation significantly outperforms the corresponding OpenMP. Additionally, the SkePU OpenMP implementation provides slightly higher performance than the original for network sizes 30K or higher, due to the fact that SkePU uses more aggressive defaults in OpenMP tuning parameters than the original implementation. Finally, experiments showed that CPU performance is lower when using less than 80 threads for the CPU implementation, for almost all input sizes. In the 10K test case, slightly higher speedup can be achieved with a lower number of threads (about 8% with 40 SkePU threads).

Fig. 11 shows the execution time of 200 time steps of three SkePU versions of the brain simulation application with 90,000 neurons and dense connectivity pattern on up to 32 nodes of the Tetralith cluster at NSC Linköping. The Inner MPI version using the most generic Mat container access proxy scales poorly, due to the excessive coherence communication volume required for this access pattern. The Outer MPI version (with hand-written MPI code and not using StarPU) has lower overhead for up to 4 nodes but shows worse scaling behavior than the Inner MPI version with the more specific MatRow container access proxy.

The technical work of applying the EXA2PRO programming model in the brain modeling application was performed by an experienced engineer who was familiar with the original application. The training time required to get familiar with SkePU was about 2 weeks and one week was required to apply SkePU. The EXA2PRO-specific LOC added to the original application were about 50.
Application developers were able to explore a variety of heterogeneous architectures, by using the single-source multi-backend feature of EXA2PRO: CO2 capture and Brain modeling were evaluated on CPU, GPU and MPI clusters, while Metalwalls on CPU, DFEs and MPI cluster. Therefore, developers were able to investigate application performance (and energy consumption in some cases, such as Metalwalls) on different computing systems and accelerators. For example, the CO2 capture application was deployed and evaluated for the first time on a GPU. The most interesting observations from the evaluation of each application on different architectures, having the corresponding native code as baseline, are shown on Table IV.

In most cases, developers used the backend implementations of computations already provided by the framework, thus improving development productivity significantly. The only exception is the MaxJ implementation in Metalwalls, which was manually provided, since MaxJ support by SkePU is still limited. However, in the CO2 capture case, the evaluation on GPU would not be initially considered a viable option, due to the very high programming effort of developing an OpenCL or CUDA port. However, after applying the EXA2PRO framework, the evaluation on GPU was performed automatically, since the SkePU already provides a GPU backend for Map skeleton (Fig. 7). (The available skeletons and the corresponding backends, already provided by the EXA2PRO framework are shown in Table II).

The data-flow programming model applied to Metalwalls provided promising results. The DFE implementation outperforms the corresponding GPU in terms of performance/watt, as shown in Fig. 9, showing that data-flow can be a relevant model for exascale computing. However, the programming effort required to port the application to a DFE is relatively high. (As mentioned in the previous section, it is almost 12 months). Reducing the amount of programming effort needed will encourage more application developers to evaluate the data-flow programming model and exploit the DFE accelerators. The EXA2PRO framework already supports DFE accelerators though the runtime system. However, support will be extended to SkePU, as well, to further increase data-flow programming model support and development productivity.

In this subsection we summarize the main observations by the analysis of evaluation results.

Application developers were able to explore a variety of heterogeneous architectures, by using the single-source multi-backend feature of EXA2PRO: CO2 capture and Brain modeling were evaluated on CPU, GPU and MPI clusters, while Metalwalls on CPU, DFEs and MPI cluster. Therefore, developers were able to investigate application performance (and energy consumption in some cases, such as Metalwalls) on different computing systems and accelerators. For example,
instantiated explicitly, for example in the libsolve ODE solver which uses BLAS1 and BLAS2 functionality.

When the EXA2PRO skeletons do not match the computational pattern of the application, multi-variant components can be developed, as explained in Section III, and/or StarPU can be directly applied. However, the programming effort in these cases is often much higher. Finally, the EXA2PRO website contains a lot of relevant training material [1] to make the framework more accessible to developers.

With respect to the complexity of the EXA2PRO application versions, evaluation results show that the use of EXA2PRO generally results in fewer LOC, since the EXA2PRO tools encapsulate the platform specific implementations. For example, the CO₂ capture, application developers reported 36-64% fewer lines code compared to manually developed CUDA, OpenCL and MPI versions of the application. Similarly, Metalwalls developers stated that EXA2PRO Metalwalls code is simpler and shorter (Section IV-B), resulting in a more maintainable codebase.

Exascale computing systems are expected to consist of heterogeneous nodes. Developers who adopted the EXA2PRO framework evaluated applications across a variety of different architectures and efficiently exploited accelerators, such as GPUs, FPGAs and more complex computing architectures compared to the ones in which they were originally deployed. Thus, EXA2PRO allows applications to grow and to exploit more complex systems with limited programming effort, contributing to the transition of applications to exascale computing.

V. CONCLUSION

The complexity of exascale systems, including heterogeneity aspects, pushes for more effective programming models. The EXA2PRO framework aims at improving application developers’ productivity, by lowering the barrier of access to exascale computing systems to the scientific community and industry. In this work, we applied the EXA2PRO framework to four applications and demonstrated how it can be used to automatically deploy and evaluate applications to a wide variety of heterogeneous clusters. Thus, the EXA2PRO framework, by hiding platform-specific details from developers, contributes to allowing more applications to move towards the direction of exascale computing systems.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their insightful comments and suggestions.

We thank Mahder Gebremedhin for help with the implementation of the Mercurium extensions, Suejb Memeti and Stavroula Zouzoula for help with the implementation of the ComPU extensions and Sotirios Panagiotou for help with the brain modeling application.

This work was supported by computational time granted from i) the National Infrastructures for Research and Technology S.A. (GRNET) in the National HPC facility - ARIS - under project EXACO2, ii) NSC Linköping and SNIC (Tetralith) under projects SNIC 2016/5-6 and SNIC 2020/13-113, and iii) PRACE (Piz-Daint) under project pr114 "EXA2PRO".

This work has received funding from the European Union’s Horizon 2020 research and innovation programme, under grant agreement No. 801015 (EXA2PRO, https://exa2pro.eu/).

REFERENCES

Christopher Kessler is a Professor for Computer Science at Linköping University, Sweden, where he leads the Programming Environment Laboratory’s research group on compiler technology and parallel computing. His research interests include parallel programming, compiler technology, code generation, optimization algorithms, and software composition/synthesis.

August Ernström is a final-year PhD student at the Department of Computer and Information Science (IDA) of Linköping University, Sweden. He received his Bachelor of Science in Computer Engineering in 2016 and his Licentiate degree in Computer Science in 2020 from Linköping University. He is the main developer of the SkePU programming framework since 2016. His main research interests are in high-level parallel programming interface design targeting multi-core heterogeneous systems and clusters. His recent work concerns how modern C++ can be leveraged and adapted to fit this purpose.

Johan Ahlqvist is a research assistant in the SkePU development team at the Department of Computer and Information Science (IDA) of Linköping University, Sweden. His research interests are in parallel computing, advanced C++ and compiler technology. He has developed the cluster backend for SkePU atop StarPU-MPI and the SkePU plug-in to the Mercurium framework.

Nikos Vasillas holds an MEng in Mechanical Engineering and an MSc in Theoretical Informatics, Systems and Control Theory from the Aristotle University of Thessaloniki. He is a Research Associate at the Chemical Process and Energy Resources Institute (CPERI) of the Centre for Research & Technology Hellas (CERTH), Greece.

Athanasios I. Papadopoulos holds an MEng in Chemical Engineering from the Aristotle University of Thessaloniki, Greece and a PhD in Process Systems Engineering from the University of Surrey, UK. He is a Principal Research Scientist at the Chemical Process and Energy Resources Institute (CPERI) of the Centre for Research & Technology Hellas (CERTH). His expertise is in the areas of sustainable materials and process design, design of reactive separation and renewable energy systems, CO2 capture and utilization processes, heat-to-power and heat-to-cooling cycles.

Panos Sefерlis is a Professor at the School of Mechanical Engineering in the Machine Dynamics Laboratory of the Aristotle University of Thessaloniki, Greece. Since 1999, he has been collaborating in various research fields at the Chemical Process and Energy Resources Institute (CPERI) of the Centre for Research & Technology Hellas (CERTH).

Lazaros Papadopoulos is a senior research associate at the Microprocessors and Digital Systems Lab of the School of Electrical and Computer Engineering of National Technical University of Athens, Greece. He received his PhD in 2015. His research interests include techniques and methodologies for memory management optimizations and high-level programming models.

Dimitrios Soudris Prof. Dimitrios Soudris received his Diploma in Electrical Engineering from the University of Patras, Greece, in 1987. He is currently working as Professor in School of ECE of National Technical University of Athens, Greece. His research interests include reconfigurable architectures, network-on-chip architectures and low-power VLSI design.
**Charles Prouveur** is a research Engineer in scientific computing at the Centre national de la recherche scientifique (CNRS), France.

**Matthieu Haefele**, PhD in computer is a CNRS research engineer since 2014 at the Laboratory of Mathematics and its Applications (LMAP) at Université de Pau et des Pays de l’Adour, Pau, France. With his expertise in HPC, he gives support to mathematicians and physicists to use efficiently supercomputers as well as producing maintainable code.

**Samuel Thibault** is an Assistant Professor at the University of Bordeaux, France. His researches revolve around task and data transfer scheduling in parallel and distributed runtime systems. He is currently focused on the design of the StarPU runtime, and more particularly its scheduling heuristics for heterogeneous architectures and for distributed systems.

**Athanasios Salamanis** is a research scientist and senior lead software engineer at the Centre for Research & Technology Hellas (CERTH). His main research interests include high performance computing, big data analytics, time series forecasting, machine learning and deep learning.

**Theodoros Ioakimidis** is a research associate at the Information Technologies Institute (ITI) of the Centre for Research and Technology (CERTH).

**Dionysios Kehagias** received his Diploma and Ph.D. degrees in Electrical and Computer Engineering from the Aristotle University of Thessaloniki, Thessaloniki, Greece, in 1999 and 2006, respectively. He is currently a Principal Researcher with the Information Technologies Institute of the Centre for Research and Technology Hellas (CERTH). His research interests include time-series analysis and forecasting, big data analytics, machine learning and algorithms for software engineering.